INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Dealing with Hardware in Embedded Software: A Retargetable Framework Based on the Devil Language

Fabrice Mérillon Gilles Muller

N°4187

Mai 2001

THÈME 2

ISSN 0249-6399 ISRN INRIA/RR--4187--FR+ENG

Dealing with Hardware in Embedded Software: A Retargetable Framework Based on the Devil Language

Fabrice Mérillon Gilles Muller

Thème 2 — Génie logiciel
et calcul symbolique
Projet COMPOSE

Rapport de recherche n ° 4187 — Mai 2001 — 15 pages

Abstract:

Writing code that talks to hardware is a crucial part of any embedded project. Both productivity and quality are needed, but some flaws in the traditional development process make these requirements difficult to meet.

We have recently introduced a new approach of dealing with hardware, based on the Devil language. Devil allows to write a high-level, formal definition of the programming interface of a peripheral circuit. A compiler automatically checks the consistency of a Devil specification, from which it generates the low-level, hardware-operating code.

In our original framework, the generated code is dependent of the host architecture (CPU, buses, and bridges). Consequently, any variation in the hardware environment requires a specific tuning of the compiler. Considering the variability of embedded architectures, this is a serious drawback. In addition, this prevents from mixing different buses in the same circuit interface.

In this paper, we remove those limitations by improving our framework in two ways. (i) We propose a better isolation between the Devil compiler and the host architecture. (ii) We introduce Trident, a language extension aimed at mapping one or several buses to each peripheral circuit.

Key-words: embedded systems, device drivers, domain-specific languages

(Résumé : tsvp)

This work has been partly supported by the French Ministry of Economy, Finance and Industry under the ITEA DESS contract 004930214, and the French Ministry of Education and Research.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Gestion du matériel dans les logiciels embarqués : une infrastructure reciblable basée sur le langage Devil

Résumé : Lors de la conception d'un système embarqué, écrire le code qui communique avec le matériel est une phase cruciale. La productivité et la qualité sont deux nécessités, mais le processus de développement traditionnel présente des points faibles qui rendent ces contraintes difficiles à satisfaire.

Récemment, nous avons introduit une nouvelle approche pour la gestion du matériel, basée sur le langage Devil. Ce langage permet d'écrire une définition formelle et de haut niveau de l'interface de programmation d'un circuit périphérique. Un compilateur vérifie automatiquement la cohérence d'une spécification Devil, puis l'utilise pour générer le code de bas niveau nécessaire au pilotage du circuit.

Dans notre première version d'une infrastructure pour Devil, le code généré est dépendant de l'architecture hôte (processeur et bus). Par conséquent, toute variation de l'environnement matériel oblige à modifier le compilateur. C'est un gros inconvénient, si l'on considère la variabilité des architectures embarquées. De plus, cela empêche d'utiliser plusieurs bus dans une même interface de circuit.

Dans cet article, nous pallions à ces limitations en apportant deux améliorations essentielles à notre infrastructure. (i) Nous proposons une meilleure isolation entre le compilateur Devil et l'achitecture hôte. (ii) Nous introduisons Trident, une extension du langage ayant pour but d'associer un ou plusieurs bus à chaque circuit.

Mots-clé : systèmes embarqués, pilotes de périphériques, langages dédiés

1 Introduction

Nowadays, most electronic appliances hitting the market, whether portable or not, are smart devices fitted with at least one CPU (microprocessor, microcontroller, or DSP) surrounded by peripheral circuits (controllers and memories). Executed by the CPU, the embedded software (or firmware) communicates with the peripheral circuits to control the behavior of the appliance.

Development issues

Depending on the complexity of the embedded software, the portion of code devoted to hardware communication may vary, but it is always a tricky part of the development:

- Contrary to the workstation world, which counts a few, mostly steady architectures, the embedded world is populated with a multitude of different CPUs, communication buses, and peripheral circuits. This variety leads to a very low rate of code reuse, all the more as embedded technologies are evolving at a frantic pace.
- Given this fast evolution, a short time-to-market is essential. Delaying a product because of firmware development could be disastrous. Consequently, once the hardware design has been settled, little time can be devoted to the development of hardware-dependent code. This can conflict with crucial quality requirements, such as reliability and robustness.
- At the lowest level, operating the hardware implies reading (writing) unstructured binary data from (to) the peripheral circuits. High-level information needs to be extracted (inserted) from (into) this raw data. Whether C or assembly is used, this leads to a chunk of low-level bitwise operations (masking and shifting) that are fairly unreadable, making such code error-prone to write and difficult to debug and maintain.
- The documentation of peripheral circuits takes the form of printable datasheets. Written in a natural language, these datasheets are often imprecise, and they may be plagued by omissions and typo errors. Incidentally, each vendor has its own vocabulary and style. Hence, interpreting the documentation is far from obvious, and the gathered information often needs to be verified experimentally.

Devil approach

Lately, we have introduced an easier and safer way of writing hardware-operating code, based on the Devil language [5, 6]. Devil is an IDL (Interface Definition Language) that allows to formally specify the programming interface of a peripheral circuit. A compiler automatically checks the consistency of a Devil specification, from which it generates all the needed low-level code, in the form of an efficient library. Embedded software is then written using the generated library. The API of the library is a functional interface of the target

circuit, not its low-level communication interface. Using Devil improves the development process in several ways:

- Devil specifications can be checked for consistency by the Devil compiler. This helps the hardware expert in writing a correct specification. This also gives a degree of confidence to the programmer who wants to reuse an existing specification.
- A Devil specification clearly documents a circuit interface, and can be considered as a knowledge repository. Once it is written, the programmer no longer has to deal with the inaccuracy of vendor's documentation. Ideally, Devil specifications should be either provided by chip vendors or available as public libraries.
- Thanks to the generated library, the programmer does not have to write the low-level code by hand. This significantly improves productivity, since it avoids a tedious and error-prone task. Moreover, Devil specifications are independent of the host CPU. They can be reused to generate code for any architecture, provided that a suitable version of the Devil compiler is available.
- Usage of the generated library can be checked for correctness, thanks to parameter types. Checking can occur at compile time and/or at run time. As shown by our recent experimental evaluation [7], this allows the early detection of many programming errors.

Devil has been proved to be expressive enough to specify a wide range of PC components, including controllers for interrupts, DMA, Ethernet, graphics, sound, ATA/IDE disks, and mice. Additionally, we have shown that using Devil does not induce any significant performance overhead [5].

Targetting Embedded Systems

When we started working on Devil, our initial goal was to ease the development of PC device drivers. In that respect, our original framework has been designed with workstations in mind. For example, we assumed a standard parallel bus between the CPU and the peripheral circuit.

As Devil proved very successful as specifying PC components, we soon realized that embedded systems would be an ideal target. The range of peripheral circuits found in embedded systems is wider than that of desktop computers, which translates into a greater need of device drivers. Thanks to the variability of embedded hardware, there are clear benefits of using Devil for the development of embedded software. However, such variability could also be an obstacle.

Since Devil specifications capture only the communication details that are intrinsic to peripheral circuits, they are independent from a particular host architecture (CPU and bus). However, the compiler must generate the architecture-specific assembly instructions that implement bus accesses. Consequently, targetting a different architecture requires a different compiler. This may be good enough for desktop computing, which counts very few

distinct architectures, but embedded applications may require the compiler to be easier to retarget.

Given its slight bias towards workstations, our original framework might not be general enough to cope easily with embedded hardware. Evaluating the extent of the problem and providing some solutions is the goal of this paper.

Contributions

In this paper, we show that the Devil language, besides being useful for PC device drivers, can be also extremely valuable for embedded software. To this end, we acknowledge the limitations of our original framework, and we present a way of extending it into a more general solution.

Our contributions can be summarized as follows:

- We analyze the existing Devil framework, and identify some aspects that do not suit the needs of embedded systems.
- We propose to better isolate the Devil compiler from the host architecture (CPU and bus), making it easier to retarget. To this end, we encapsulate the architecture-dependent code into *Bus Abstraction Modules*.
- We introduce *Trident*, an extension of Devil aimed at mapping a bus architecture to each peripheral circuit.

The rest of this paper is organized as follows. Section 2 briefly presents the Devil language. Section 3 discusses the problems that can be faced when using Devil for embedded systems. Section 4 and 5 show how these problems can be solved. Section 6 describes related work. Section 7 concludes the paper.

2 Devil in a Nutshell

Devil is an IDL aimed at specifying the programming interface of a device controller. From a Devil specification, a compiler generates a library containing all the low-level code that is required to operate the device. The generated library is then used to write the higher-level code of the device driver.

To design Devil, we have studied a wide spectrum of PC components: controllers for Ethernet, display, sound, disk, interrupts, DMA and mouse. This study was supported by driver source code (mainly from Linux), books about driver development [2, 8], device documentation available on the web, and discussions with device driver experts (for Windows, Linux and embedded operating systems).

```
device saa1064 (base : portbase)                                                    1
{
   ports base @ {0..4} : bit[8];                                                    3

   register control_reg = base @ 0, mask '0.......' : bit[8];                       5

   variable display_mode = control_reg[0] : { STATIC <=> '0', DYNAMIC <=> '1' };    7

   structure dynamic_control = {                                                    9
      variable digit1_digit3_enabled = control_reg[1] : bool;                      10
      variable digit2_digit4_enabled = control_reg[2] : bool;                      11
   };

   variable all_leds_on = control_reg[3] : bool;                                   14

   structure output_current = {                                                    16
      variable add_3mA  = control_reg[4] : bool;                                   17
      variable add_6mA  = control_reg[5] : bool;                                   18
      variable add_12mA = control_reg[6] : bool;                                   19
   };

   register digit1_reg = base @ 1 : bit[8];  variable digit1 = digit1_reg : int(8);   22
   register digit2_reg = base @ 2 : bit[8];  variable digit2 = digit2_reg : int(8);   23
   register digit3_reg = base @ 3 : bit[8];  variable digit3 = digit3_reg : int(8);   24
   register digit4_reg = base @ 4 : bit[8];  variable digit4 = digit4_reg : int(8);   25
}
```

Figure 1: Specification of Philips' SAA1064 (4-digit LED Display Controller)

Language Features

Concretely, the programming interface of a peripheral circuit can be described by three main abstractions: *ports*, *registers*, and *device variables*. The entry point of a Devil specification is the declaration of the *device*, parameterized by one or more *port bases*, from which ports are derived. Ports, which abstract physical addresses, allow registers to be declared. Registers define the granularity of interactions with the device. Finally, device variables are defined from registers, forming the functional interface of the device. Thus, a Devil specification is structured into three layers: one layer per main abstraction. We now present each layer, taking the example of the Devil specification of Philips' SAA1064, a controller for LED displays (see Figure 1).

Ports A port abstracts a physical address, that is a communication point between the peripheral circuit and the CPU. A device often features several ports whose addresses are derived from one or more base addresses. In Devil, a base address is always a parameter of the specification, and gets the type `portbase`. The port constructor, denoted by `@`, takes as arguments a `portbase` and a constant offset (*e.g.,* `base @ 0` as illustrated by line 5 of the

SAA1064 specification). To enable verification, the range of valid offsets must be specified using the `ports` keyword, as illustrated by line 3 of the SAA1064 specification.

Registers Registers define the granularity of interaction with a device. Thus, the size of each register (number of bits) must be explicitly specified. Registers are typically defined using two ports: a read port and a write port. Only one port needs to be provided when reading and writing share the same port, or when the register is read-only or write-only.

A register declaration may be associated with a mask that specifies bit constraints. An element of this mask can be either: '`.`' to denote a used bit, '`0`' or '`1`' to denote an unused bit that is undefined when read but has a fixed value (`0` or `1`) when written, or '`*`' to denote a unused bit that is always undefined. As an example, consider the declaration of the register `control_reg` at line 5 of the SAA1064 specification. The mask indicates that bit 7 (the leftmost bit) is not used but must be written as 0.

Device variables In order to minimize the number of I/O operations required for communicating with a device, hardware designers often group several independent values into a single register. Accessing these values requires bitwise operations (masking and shifting), whose code is error-prone to write in a general programming language such as C. Devil abstracts values as device variables, which are defined as a sequence of register bits. Device variables are strongly typed in order to detect potential misuses of the device. Possible types are booleans, enumerated types, signed or unsigned integers of various sizes, and ranges or sets of integers. At line 14 of the SAA1064 specification, bit 3 of the `control_reg` becomes a boolean variable.

And more... Many features of Devil are not detailed here. These features include indexed registers, enumerated types, register concatenation, structures, arrays, and behavioral attributes. The language is described into more details in our previous papers [5, 6], as well as in the reference manual [9].

3 Applying Devil to Embedded Systems

Although the benefits of using Devil for embedded software seem obvious, doing it in practice may reveal some limitations of our original framework. Since Devil was originally designed to ease the development of PC device drivers, its ability to address the peculiarities of embedded hardware has yet to be shown. This section digs into on this question.

Variability of Embedded Hardware

Although some embedded systems have a hardware architecture similar to that of any desktop computer, constraints of cost and size often lead to less conventional designs. In addition,

being freed from software compatibility issues (firmware is tailored to specific hardware) allows a faster evolution than in desktop computers. Ultimately, system designers have a wide range of hardware solutions to choose from. We now enumerate in what aspects embedded architectures can depart the most from desktop architectures, as far as Devil is concerned.

Buses Both desktop and embedded systems make use of parallel buses. They can vary in width, being multiplexed or not, etc. One difference however, is that embedded buses are much more likely to change from one system to another. Such variety can have consequences on the hardware-operating code, like several sizes for `port` addresses. Indeed, properly typing such addresses would require to generate a different code for each size.

Besides, a notable peculiarity of embedded systems is their wide use of serial buses, such as Philips' ubiquitous I^2C standard. Workstations have started to follow this trend, with the famous USB (Universal Serial Bus), the awaited Serial-ATA (for internal hard drives), and various derivatives of I^2C such as the SMBus (for motherboard sensors) and the DDC (Display Data Channel, for monitor configuration) [13, 10, 11, 1]. With I^2C, selecting a `port` requires two address values (a chip ID and a register index), instead of one for parallel buses. Once again, handling such a peculiarity requires specific code.

CPUs Basically, an embedded CPU is either a microprocessor, a DSP (Digital Signal Processor), or a microcontroller. Besides their signal processing orientation, DSPs are actually microprocessors [4]. Typically, a microprocessor is fitted with a parallel bus interface, which rely on a simple, hardware-implemented bus protocol. Very different in that respect, a microcontroller is fitted with a more versatile interface, made of general-purpose I/O pins. Software can control each of these pins individually, set it for input (output), and read (write) its logical state. This allows bus protocols to be implemented in software. Consequently, microcontrollers can emulate various bus interfaces, whether parallel or serial.

In addition, each family of CPU has its own instruction set. As long as software is written in C this is not a problem: source code is portable and the variability is hidden. However, hardware-operating code needs to make direct accesses to I/O buses. Since that kind of operation is not abstracted by the C language, the only way of accessing buses is to use assembly. In other words, hardware-operating code always contain a part of CPU-dependent code. Such part is typically limited to a few dedicated read/write instructions (such as IN/OUT for x86 microprocessors, or BTFSC/BCF for PIC microcontrollers). This is much more an issue in embedded systems, which count more intruction sets than desktop computers.

Limited Retargetability

A conclusion of the previous subsection could be that retargetability is a key feature when generating code for embedded sytems. In our existing Devil framework, the CPU-dependent code (I/O instructions) is supplied by Linux, in the form of small macros containing assembly code. Thus, replacing this set of macros could be sufficient for retargetting the Devil compiler

```
 int read (char *device, char *portbase, int offset, int size);
void write(char *device, char *portbase, int offset, int size, int data);

void block_read (char *device, char *portbase, int offset, int size, int iterations, void *data);
void block_write(char *device, char *portbase, int offset, int size, int iterations, void *data);

void range_read (char *device, char *portbase, int offset, int size, int iterations, void *data);
void range_write(char *device, char *portbase, int offset, int size, int iterations, void *data);
```

Figure 2: Common API for Bus Abstraction Modules

to a different CPU. In the case of a microcontroller handling the bus protocol in software, those macros could contain all the protocol implementation.

However, this way of retargetting Devil would require the bus architecture not to change. This may be the case for x86-based of Alpha-based workstations, but not for embedded systems. Indeed, as soon as the bus changes, the size and format of `port` addresses change too. To handle this, various small details must be modified in the Devil compiler, which is akward. Moreover, this does not solve the problem of having several bus interfaces in the same chip. National Semiconductors' LM78 is an example: it features both an ISA and an I^2C interface. Because one Devil compiler cannot handle both bus types, the only solution is to split LM78's programming interface into two distinct Devil specifications.

Finally, when applying Devil to embedded systems, the main challenge is to deal with the variety of bus architectures. In the following two sections, we show how to improve the retargetability of Devil.

4 Bus Abstraction Modules

In the previous section, we saw that accessing buses requires a few assembly macros containing CPU-specific I/O instructions. In a sense, such macros constitute a library, that abstracts the bus/CPU combination and provides communication services to the higher-level C code. To generalize this idea, such a set of macros will now be called a Bus Abstraction Module (BAM).

Single BAM, single API. In modern workstations, whatever the CPU, bus architectures are typically based on PCI. This uniformity allows all BAMs to share the same API. Thanks to this API, the hardware-operating code can be source-portable accross CPUs. In Linux and Windows, the BAM is part of a more general Hardware Abstraction Layer (HAL), that abstracts most of the CPU-dependent parts of the OS kernel. The C code generated by our existing Devil compiler makes use of the Linux HAL. The API consists of the so-called "I/O calls": `inb`, `outb`, `inw`, `outw`, etc.

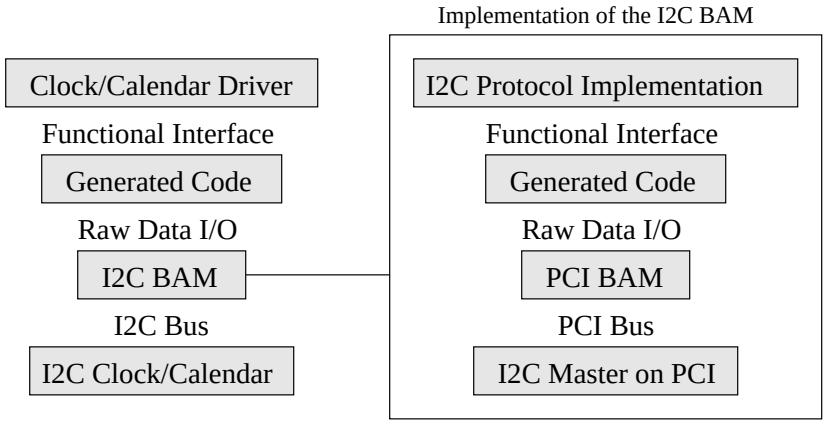

Figure 3: Using Devil to implement BAMs

Multiple BAMs, multiple APIs. In embedded systems, the variety of bus achitectures prevents from using the same API for every BAM. For example, a serial I^2C bus does not use the same address format than a PCI bus. In our original framework, targetting a different BAM means generating a different code. This requires the compiler to be modified, which is rather awkward. Moreover, this prevents from targetting two different BAMs in the same circuit interface.

Multiple BAMs, single API. A solution for removing these limitations would be to use a single API for all BAMs. As said above, differences in the address format prevent from doing so. A solution would be to use portbase names instead of addresses, and to associate portbases with addresses in a separate configuration file.

As each BAM provides the same services, that is reading (writing) raw data from (to) the bus, the main primitives of the generic API should be `read` and `write`. However, these primitives should support single accesses, block accesses (several consecutive accesses with the same address), as well as range accesses (several consecutive accesses with an auto-incremented address). In addition, all primitives should provide a parameter for selecting the bus (i.e., the underlying BAM). The name of the portbase is sufficient, provided that the mapping between portbases and BAMs is specified somewhere else. Once again, the solution of a separate configuration file seems adequate. The function prototypes of a suitable API are presented on figure 2.

Stackable BAMs. An advantage of having the BAMs separated from the compiler, is that they become stand-alone components. Thus, it becomes easier for the programmer to write its own BAMs. Another benefit is the possible stacking of BAMs. Indeed, nothing prevents from using a BAM to write another (see figure 3). This extends the naive model

of one, monolythic physical link into the more realistic model of a physical link made of several buses. For instance, when the CPU talks to an ATA/IDE disk, data goes through the CPU local bus, the PCI bridge, the PCI bus, the ATA interface, and the ATA bus. This is transparent, and entirely handled by the hardware, so the normal PCI BAM does the job. However, in the case of an I^2C device attached to a PC through a PCI adapter, a special I^2C BAM needs to be written, using Devil to specify the I^2C master (the bridging circuit) present in the PCI card. This is the example shown on figure 3.

5 Trident

Thanks to the BAMs, Devil can target several buses with the same compiler.

In this section, we present Trident, an extension to the Devil language aimed at providing each portbase with a BAM and a bus address.

Port Base Instanciation

In Devil, a `port` is always given a relative address, that is a qualifying number that identifies the `port` relatively to a given `portbase`. This can be compared to a file name, that identifies the file relatively to a given directory. The absolute address of the `portbase` (comparable to the absolute path of a directory) is not part of the circuit interface. The address format depends of the bus architecture of the host system, and the value of the `portbase` might not be known before run-time. Consequently, the Devil language allows only the declaration of portbases. They are not provided with any definition or value. In that respect, a Devil specification is like a template that needs to be instanciated. This is not surprising, since a Devil specification is not supposed to assume a particular use for the chip it specifies. Once a circuit is used in a particular application, its hardware environment is known and the portbases can be instanciated.

Instanciating portbases means providing them with a bus and an address. A set of Trident declarations is a kind of configuration file which provides such information.

Trident Syntax

Let us start with an example. Figure 1 shows the Devil specification of a LED display controller, which is fitted with an I^2C interface. Such a peculiarity becomes obvious when looking at the following Trident fragment:

```
portbase base = i2c_7 {
   chip_id = dynamic : {113, 115, 117, 119};
   reg_index = 0 : {0..4};
}
```

The first line is the most important: it binds the `portbase` named `base` to the BAM named `i2c_7` (I^2C bus with chip IDs limited to 7 bits). The BAM identifier will be used by the compiler to guess in what source files the BAM is implemented. Lines 2 and 3 define

the address of the `portbase`. As already said, this is a two-part address. Here is the general syntax for each part:

```
<identifier> = <value> : <type>;
```

In the `chip_id` definition, the `dynamic` keyword means that the exact value will only be selected at run-time. As for the type, it specifies the four possible IDs of that particular chip. The `reg_index` definition is now easy to understand: it takes the value 0 and must stay in the range 0 to 4 (5 registers). This is consistent with line 1 of the Devil specification.

We can now have a look at the Trident declaration of the I^2C BAM:

```
bus i2c_7 {
   address : {
        chip_id : int(7);
      reg_index : int(8);
   };
   data : { bit[8]; };
};
```

The second line, with the `address` keyword, specifies the format of an I^2C address. Inside the curly brackets, we find the usual two parts, with their types. The `data` keyword specifies all possible data sizes of the bus (only 8 bits in this example).

Development Process

Figure 4 give some details on how Trident contributes to the development process:

- Trident declarations, kept in a text file, are provided the Trident compiler.
- The Devil compiler, once the Devil specification is parsed, produces a text file containing typing information about portbases and ports. This file is provided to the Trident compiler.
- The Trident compiler parses the Trident declarations, and checks that the portbase declaration is consistent with the bus declaration and with the information provided by the Devil compiler.
- Bus abstraction Modules that are used in the Trident declarations are provided to the Trident compiler.
- The Devil compiler generates code, using a generic, architecture-independant API (see figure 2) wherever a bus access is needed.
- The Trident compiler browses the generated code, and replaces each generic call by an instanciated version of the bus access code found in BAMs. The second parameter of each generic call, `portbase`, refers to a portbase declaration where the Trident compiler can find the name of the adequate BAM. This preprocessing of the generated code is the main task of the Trident compiler.

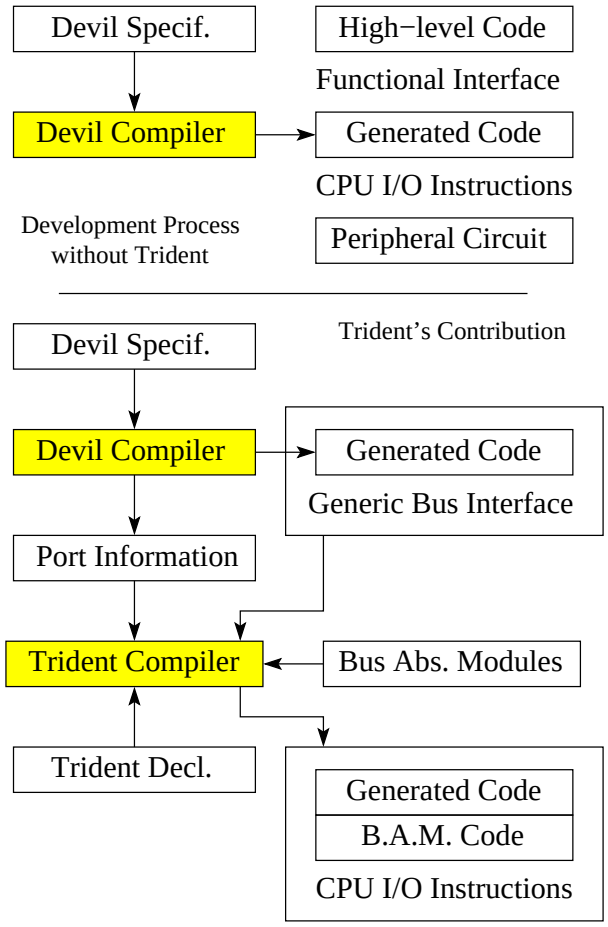

Figure 4: Trident in the Development Process

- The code generated by the Devil compiler now targets the bus architecture specified in Trident declarations.

6 Related Works

The initial idea of Devil comes from earlier work at the Compose Group. Thibault & al [12] designed a language, called GAL, aimed at specifying X11 drivers (for graphics adapters).

Although successful as a proof of concept, GAL covered a very restricted domain.

In previous studies, we described the Devil approach in details [5, 6]. Additionally, we showed that using Devil to develop device drivers did not induce any significant performance overhead [5]. More recently, we conducted a thorough experimental evaluation of Devil benefits over the traditional development process, in terms of productivity and code robustness. [7]

Languages for specifying digital circuits and systems have existed for many years. The VHDL standard [3], widely used in this domain, is one of the most expressive. It addresses several aspects of chip design such as documentation, simulation and synthesis. VHDL provides both high-level and low-level abstractions: arrays and loops are supported, as well as bit-vector literals and bit extraction. However, all VHDL abstractions focus on the inner workings of circuits, not their high-level programming interface. As a consequence, chip interfaces are not explicitly denoted, and VHDL compilers perform limited consistency checks. Interestingly, VHDL allows attaching arbitrary strings to variables. Using them to add interface-specific information is possible, but would require a normalized syntax and compiler support, which in some way amounts to embedding Devil concepts into VHDL.

7 Conclusion

Devil is a language aimed at specifying the programming interface of device controllers. From the Devil specification of a controller, a compiler automatically generates a library that contains all the low-level code needed to operate the controller. The generated library can then be used to write higher-level software, such as a PC device driver or the firmware of an embedded system.

In this paper, we have shown that Devil could ease the development of embedded software. To this end, we spotted the weaknesses of our original framework, and provided some solutions.

As a matter of fact, the improvements described in this paper are not only useful for embedded systems. They could also help writing some PC device drivers that were out of reach with the original framework. For example, most recent PCs have several captors (temperature, fan speed, etc.) connected to the motherboard chipset through the SMBus [11] (a simplified version of I^2C). Because these captors are not directly connected to the CPU, supporting them would have required a dedicated version of the Devil compiler. The ability to stack BAMs, like on figure 3, provides an elegant mean of supporting the SMBus.

Although Trident declarations are made necessary by the addition of BAMs in the Devil framework, they are also very valuable as a documentation source. The communication path between the peripheral circuits and the CPU was only partially specified by the Devil language. Trident fills the gap, providing useful information on how a device is "glued" into the host architecture.

Availability

The Devil compiler and some related material are available at the following web page: `http://compose.labri.fr/prototypes/devil`.

References

[1] Display Data Channel Command Interface (DDC/CI). Web site. http://www.vesa.org/.

[2] E. N. Dekker and J. M. Newcomer. *Developing Windows NT device drivers : A programmer's handbook*. Addison-Wesley, first edition, March 1999.

[3] IEEE Standards. *1076-1993 Standard VHDL Language Reference Manual*, 1994. http://standards.ieee.org/.

[4] M. Levy. Microprocessor and DSP technologies unite for embedded applications. *EDN Magazine*, Issue 5, March 1998. http://www.ednmag.com/.

[5] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, October 2000.

[6] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. A DSL Approach to Improve Productivity and Safety in Device Drivers Development. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000. IEEE Computer Society Press.

[7] L. Réveillère and G. Muller. Improving Driver Robustness: an Evaluation of the Devil Approach. In *Proceedings of the 2nd International Conference on Dependable Systems and Network (DSN 2001)*, Göteborg, Sweden, July 2001.

[8] A. Rubini. *Linux Device Drivers*. O'Reilly, first edition, February 1998.

[9] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. The Devil Language. Technical Report RT-0244, INRIA, Rennes, France, October 2000.

[10] Serial ATA. Web site. http://www.serialata.org/.

[11] System Management Bus (SMBus). Web site. http://www.smbus.org/.

[12] Scott Thibault, Renaud Marlet, and Charles Consel. A domain-specific language for video device driver: from design to implementation. In *Proceedings of the 1st USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[13] Universal Serial Bus (USB). Web site. http://www.usb.org/.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399